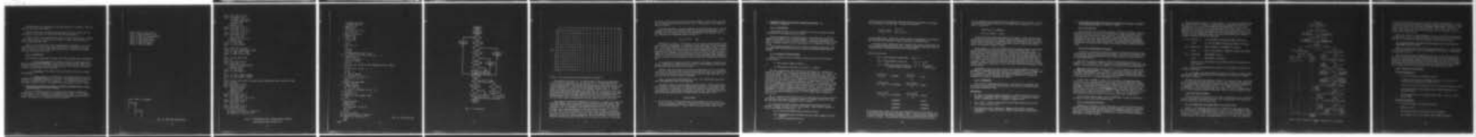
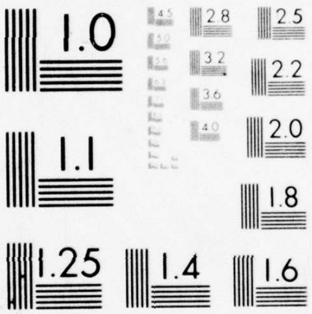
AD-A036 721 POLYTECHNIC INST OF NEW YORK BROOKLYN DEPT OF ELECTR--ETC F/G 9/2
SUMMARY OF TECHNICAL PROGRESS, SOFTWARE MODELING STUIDES.(U)
JAN 77 M L SHOOMAN, H RUSTON F30602-74-C-0294
UNCLASSIFIED POLY-EE/EP76-013 RADC-TR-76-405 NL

1 OF 1
AD
A036721



END
DATE
FILMED
4-77



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

12

RADC-TR-76-405
Technical Report
January 1977

SUMMARY OF TECHNICAL PROGRESS, SOFTWARE MODELING STUDIES

Polytechnic Institute of New York

Approved for public release;
distribution unlimited.

ROME AIR DEVELOPMENT CENTER
AIR FORCE SYSTEMS COMMAND
GRIFFISS AIR FORCE BASE, NEW YORK 13441

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public including foreign nations.

This report has been reviewed and is approved for publication.

APPROVED: Alan N. Sukert
ALAN N. SUKERT, Capt, USAF
Project Engineer

APPROVED: Robert D. Krutz
ROBERT D. KRUTZ, Col, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss
JOHN P. HUSS
Acting Chief, Plans Office

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(19) REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

(18) ~~1. REPORT NUMBER~~
RADC-TR-76-405

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
(6) SUMMARY OF TECHNICAL PROGRESS, SOFTWARE MODELING STUDIES,

(9) ~~5. TYPE OF REPORT & PERIOD COVERED~~
Interim Report,
1 Jan [redacted] - 30 Jun 76,

~~6. PERFORMING ORG. REPORT NUMBER~~
(14) Poly-EE/EP76-013

~~7. AUTHOR(s)~~
(10) M. L. Shooman
H. Ruston

~~8. CONTRACT OR GRANT NUMBER(s)~~
(15) F30602-74-C-0294

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Polytechnic Institute of New York
333 Jay Street
Brooklyn NY 11201

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
62702F
55500806

11. CONTROLLING OFFICE NAME AND ADDRESS
Rome Air Development Center (ISIS)
Griffiss AFB NY 13441

~~12. REPORT DATE~~
(11) Jan [redacted] 77

13. NUMBER OF PAGES
36

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)
Same

(12) 31p.

15. SECURITY CLASS. (of this report)
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

~~16. DISTRIBUTION STATEMENT~~ (of this Report)
Approved for public release; distribution unlimited.

(16) 5550 (17) 08

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES
RADC Project Engineer: Capt Alan N. Sukert (ISIS)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software Modeling	Programming Techniques
Software Errors	Software Reliability
Program Testing	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

During the period of time 1 Jan 76 to 30 Jun 76, Polytechnic Institute of New York conducted research under RADC contract F30602-74-C-0294 in the area of software reliability. This report presents the progress of this research. Subjects of investigation were error generation and seeding/tagging models, measures for the evaluation of software, analytical data selection methods for program testing, modular programming techniques, methods for finding feasible program paths, statistical program testing and proving, and methods

over

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

408 717

1/B

for automatically testing every program path.

Work has been completed on development of error generation/manpower deployment models to describe the error correction process in terms of error generation and correction rates as well as the number of man-months spent in debugging. Work has also been completed in the development of seeding/tagging techniques to estimate the number of software errors and related statistical quantities; measures for comparing programs, such as accessibility, testability, and testedness, based on defining a program as a set of executible modules; and a method for selecting test data sets for a program based on determining the interrelationships among program variables.

Work still in progress includes development of techniques to interactively write programs using stored library modules and/or user supplied code, investigation of a satisfactory algorithm to estimate the number of feasible paths in a program, development of a statistical theory for program testing and proving based on using a strategy of both testing and journal proving using Black's model, and implementation of a PL/1 driver to automatically test every possible path and catch errors of any PL/1 program, subject to some minor constraints.

SECTION I

INTRODUCTION

This interim report summarizes the effort expended from January 1, 1976 to June 30, 1976 on RADC Contract No. F-30602-74-C-0294. The major topics investigated and participating personnel during this period are listed in Section II. Summaries of these topics appear in Section III. Section IV indicates the planned direction of the research for the immediate future. Section V reports on the professional activities of the staff during the reporting period.

SECTION II

PERSONNEL AND WORK AREAS

The following personnel participated in the research activities during this reporting period

M. Shooman
H. Ruston

M. Adamowicz
D. Baggi
S. Habib
A. Laemmel
E. Lipshitz

S. Mohanty
S. Natarajan
G. Popkin
B. Rudner

and worked in the following areas:

1. Shooman and Natarajan:

Prediction models for estimating the number of errors in computer software. The models incorporate the error generation present during the debugging process. The analysis also compares the cost of debugging with the cost of rewriting for software in which the error removal proceeds at a small rate.

2. Shooman and Popkin:

Continuation of the work on establishing feasible program test paths.

3. Shooman and Lipschitz:

Further work on automatic programming techniques for the construction of reliable programs.

4. Ruston:

Planning of tests for the collection of data to verify the theoretical studies, and to collect statistics needed for model parameters.

5. Adamowicz and Mohanty:

Completion of two studies, one on quantitative software measures and the other on the selection of program test data.

6. Laemmel:

Completion of the study on hierarchies of computable functions with application to determination of program complexity. Continuation of the work on probabilistic models of program testing.

7. Rudner:

Completion of studies on the seeding and tagging estimates of the initial number of program bugs.

8. Baggi

The construction of a driver program for verification of Shooman's model of an exhaustive test.

SECTION III

SUMMARY OF PROGRESS

In this section we summarize the work performed. Upon completion of each task, a report on the task's results will be issued. Several technical reports which document either the completed or continued research are in various stages of preparation.

3.1 Effects of Manpower Deployment and Error Generation on Software Reliability - by M. Shooman and S. Natarajan

This phase of the modeling effort has been completed, and the detailed technical report is in typing. The abstract of the report is presented below.

Abstract

Several previous models in the literature have discussed how the number of errors in a large software system is related to the rate of error removal. Similar probabilistic models for the error removal rate were proposed in 1971 in two papers, one authored by Shooman and the other authored by Jelinski and Moranda. The models proposed by Shooman were based on error data on 7 different large operating systems and application programs and collected by Hesse, and these models also fit the data of Akiyama which was collected on small programs. Expressions for the number of remaining errors as the software undergoes debugging were formulated and additional assumptions were made to relate the number of residual errors to the operational system reliability.

A key assumption in the above models is that the sum of the errors removed and the errors remaining in the program is a constant. Thus, if we can estimate the initial number of errors in the system at the start of debugging and keep careful records of those removed we have a good estimate of the number of remaining errors. In 1973 Shooman described a test procedure for estimating the initial number of errors.

In this work we add a major refinement to the above models by introducing the possibility of error generation during debugging. A generated error is due to one of two causes: (1) a bug whose correction is invalid and further debugging on the same statements is essential, (2) a new bug which is generated as the result of the correction of a different error. The error generation terms are modeled in several different ways: proportional to the number of detected errors, corrected errors, the number of remaining errors, or some function of these effects. The correction rate is assumed to be a function of the manpower deployed on the project, thus, one can use the model to investigate optimum manpower deployment strategies. The effects on the economics of debugging due to error growth have also been analyzed.

3.2 Seeding/Tagging Estimates of the Number of Software Errors - by Beulah Rudner

The theoretical part of this effort has been completed and the technical report is in typing. The tests for experimental verification are now being planned. The abstract of the technical report is presented below.

Abstract

Seedling/tagging estimates of the number of software errors are computed from s, t and c where: t is the number of errors either inserted deliberately in a program (seeded) or found by debugging (tagged); s is the number of errors found by a debugger unaware of the contents of the first set; and c is the number of errors appearing in both sets.

Two types of questions can be raised. One type relates to the method and procedure: the introduction of new errors, the changing of a program by debugging, etc. The other relates to possible estimates, and their evaluation and comparison. This report concerns itself with questions of the second type. Estimates based on 3 models are discussed. The models are defined by assumptions regarding the equal or unequal difficulty of uncovering individual errors. Model 1 assumes all errors are equally open to discovery at all times. Models 2 and 3 assume categories of difficulty to be defined for all programs. In Model 2 the error distribution among categories is unknown; in Model 3 it is known. Estimates for Models 2 and 3 are shown to be closely related to those for Model 1.

The mean and mean - squared error of a maximum likelihood estimate and a modified maximum likelihood estimate are given. It is shown how these quantities vary with certain relations among the total number of errors, size of tagged or seeded set and size of accompanying sample set. Curves are drawn which can be used to determine optimum values for s and t and a procedure is outlined for doing so.

More precise estimates can be obtained with several trials rather than one as described above. Several such estimates are examined and discussed.

It is concluded in general terms that a reasonable investment of time will produce adequate estimates.

3.3. Measures for the Evaluation of Software - by S. N. Mohanty and M. Adamowicz

This effort has been completed and the final technical report is being prepared for typing. The abstract of the report is presented below.

Abstract

In this report an attempt has been made to define and quantify the structural qualities of computer programs for the purpose of comparing program structures. Three software quality measures (accessibility, testability, and testedness) have been developed and applied to determine how well a program has been tested.

A program is defined as a set of executable modules which are either segments or nodes; a node being a decision point and a segment being a sequence of executable statements. The accessibility of each of these modules is defined in terms of the probability of accessing the module from the previous modules and the probability of successful execution of the previous modules. The testability of each executable module is a function of the accessibility and the complexity measure of the module. The testability of a logical path is given as an average of the testabilities of the program modules constituting the path, and the program testability is given as an average of the testabilities of the paths constituting the program. The measures provide a quantitative means of comparing programs.

3.4. An Analytical Data Selection Method for Program Testing - by S. N. Mohanty and M. Adamowicz

This work has been completed and the final technical report is being prepared for typing. The abstract of the report is shown below.

Abstract

A method is presented for determining the interrelationships among variables in a program. These interrelationships provide the constraints for the input space, i.e., the space formed by the set of all possible data sets. Under this method a program statement is broken up into many basic statements. The set of all variables which consist of the set of all primary or independent variables and secondary or dependent variables are found, and the initial constraints for the primary variables are determined. The interdependencies among the variables are then tabulated in the form of a matrix called a Dependency Matrix (DM). The initial constraints for the variables are then determined locally, i.e., from the information in each basic statement and are tabulated in the form of a matrix named an Initial Attribute Matrix (IAM). To determine the dependencies among the variables, we modify the IAM and arrive at the modified IAM.

A set of algorithms have been developed for analyzing programs with arithmetic operators (i.e., **, *, +, -, /) and certain functions, namely SORT, LOG, and EXP, so as to determine the numerical bounds of the variables which are forced by the use of these operators of functions. From the DM and modified IAM, we find the numerical bounds for primary variables by making use of these algorithms.

This forms the desired input-space from which the data sets can be selected for testing the program.

3. 5. Modular Programming Techniques - by Elan Lipshitz

3. 5. 1. Introduction

The objective of this effort is to find ways to write more reliable programs. To this end we raise questions in the following three areas:

1. Computer Language Can we develop a high level language that is less susceptible to bugs? Can we prove that any one of the existing languages is better in the sense of having less bugs?

2. Structural Programming and Complexity of Programs Can we derive a sequence of steps in writing programs that will result in more reliable programs? Is there a correlation between the structure and content of a program and the number of bugs in it? If yes, which areas or statements or programming techniques are more bug-manifested? Can we eliminate or replace them?.

3. Automatic Programming Will the use of pre-written and tested modules of code reduce the number of bugs? How can they be best incorporated into the programs?

This project will investigate both structural and automatic programming with the emphasis on the latter.

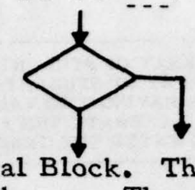
3. 5. 2. The Program: Auto - Programming

The program "Auto-Programming" is divided into two parts - "Flow" and "Auto," both of which are interactive on-line programs that, by communicating with the user, generate his program. Currently, they are written and generate programs in Fortran.

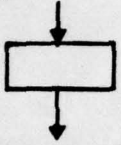
A. The Program Flow

"Flow" receives the information about the flow-chart of a program from the user. "Flow" recognizes only four different types of blocks which are sufficient to generate any flow-chart. They are:

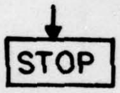
1. Control Block. This is a conditional decision block, similar to the statement IF ( ) GO TO ___. The user will write the code for this block.

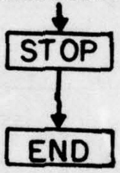


2. Functional Block. This block will perform a task that is available in the computer library. The program "Auto" will generate the correct code for this block.



3. Stop Block. This block indicates the end of the path; and is coded by the STOP statement.





4. User's Code Block. The user inserts the desired code into this block. This block is used whenever the library cannot perform the needed task.

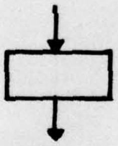


Fig. 1 illustrates how the above blocks can be used to construct the flow chart of a given program. The program first builds a table "Clas" with the names of the students in a class, and then enters their grades into the table "Grad." Note that the GO TO A is a user's code block (i.e., type 4).

The flow chart generated using only the above blocks is always a binary tree in structure, while the transfer of control during the execution might behave like a graph.

Upon completing the flow-chart control passes to the program "Auto", to be discussed next.

B. The Program: Auto

The user will specify to "Auto" what he would like to do, and "Auto" will advise him which methods are available for the solution, as well as their characteristics. The user then will choose the method he prefers, and "Auto" will generate the needed code.

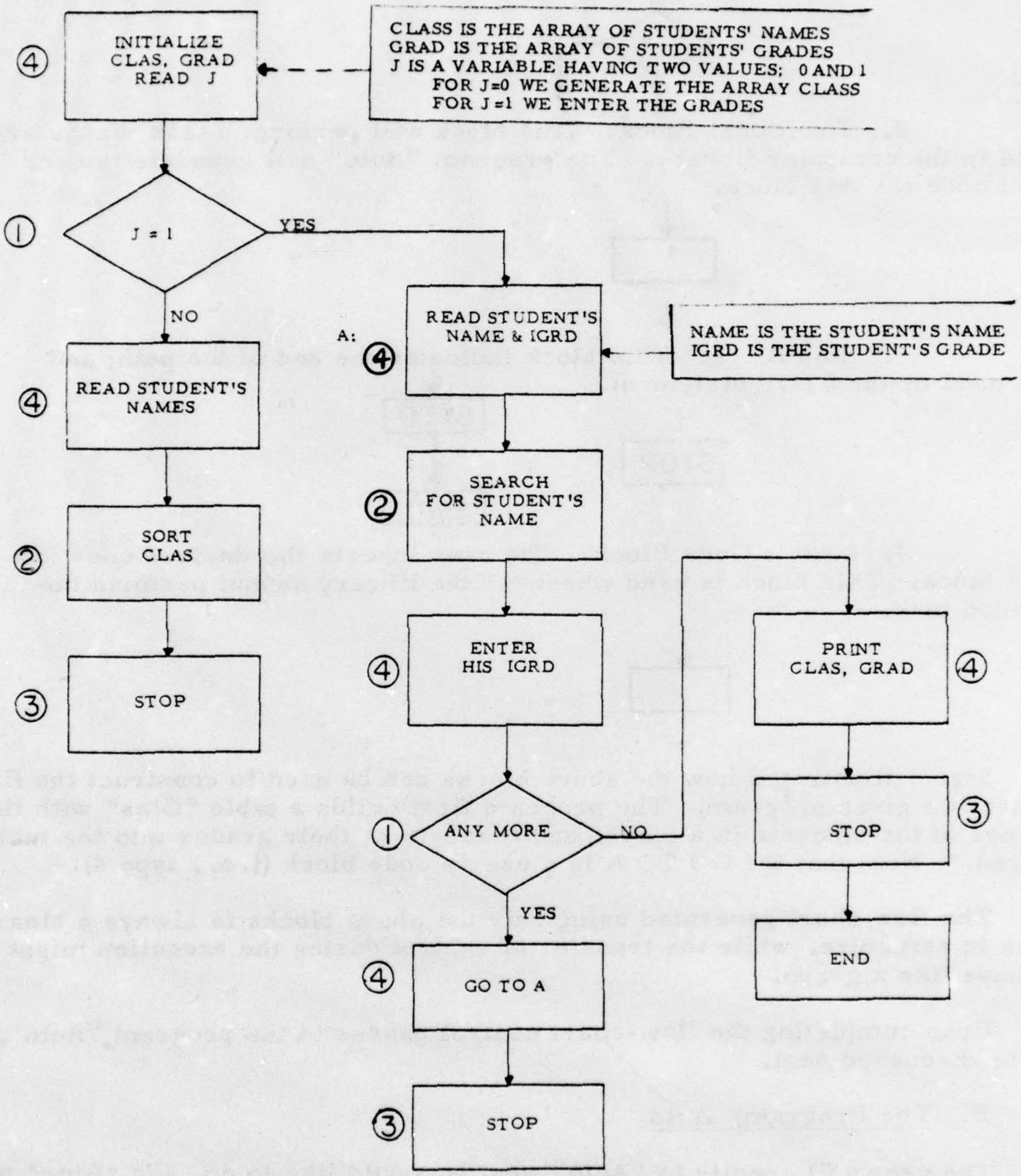

④
INITIALIZE
CLAS, GRAD
READ J
CLASS IS THE ARRAY OF STUDENTS' NAMES
GRAD IS THE ARRAY OF STUDENTS' GRADES
J IS A VARIABLE HAVING TWO VALUES; 0 AND 1
FOR J=0 WE GENERATE THE ARRAY CLASS
FOR J=1 WE ENTER THE GRADES
①
J = 1
YES
NO
A:
④
READ STUDENT'S
NAME & IGRD
NAME IS THE STUDENT'S NAME
IGRD IS THE STUDENT'S GRADE
④
READ STUDENT'S
NAMES
②
SEARCH
FOR STUDENT'S
NAME
②
SORT
CLAS
④
ENTER
HIS IGRD
PRINT
CLAS, GRAD
④
③
STOP
①
ANY MORE
NO
③
STOP
YES
④
GO TO A
END
③
STOP

Fig. 1: Record of Student's Grades

"Auto" is divided into four parts:

1. Dictionary, which advises the user of tasks available in the computer library.

2. Communicant, which supplies and receives from the user all the information needed to generate the code. In addition, it will sort the labels generated by the user.

3. Code generator, which uses information obtained with the help of the communicant, and generates the final code. It will resolve any conflicts in the labels generated by the user, and the duplicates generated by "Auto."

4. Library, which contains a collection of code modules needed for the different tasks, e.g., input/output, solution of mathematical problems, inventory and banking programs, etc.

"Auto" was revised to allow dynamic label generation. The communicant scans the user's statements looking for labels. The labels found are stored in a table "Lab" in descending order. Before the code generator writes a label it searches "Lab" to see whether this label is already in use.

The library is being expanded. Code has been written and tested for the following mathematical tasks:

$$L_n(x) = 2 \sum_{n=0}^{m} \frac{1}{2n+1} \left(\frac{x-1}{x+1} \right)^{2n+1} \qquad .5 \le x \le 2.0 \qquad (1)$$

$$\ell^x = \sum_{n=1}^{m} \frac{x^n}{n!} \qquad -2.0 \le x \le 2.0 \qquad (2)$$

$$\text{Sin}\ (x) = \sum_{n=0}^{m} \frac{(-1)^n\ x^{2n+1}}{(2n+1)!} \qquad -2\pi \le x \le 2\pi \qquad (3)$$

$$\text{Cos}\ (x) = \sum_{n=0}^{m} \frac{(-1)^n\ x^{2n}}{(2n)!} \qquad -2\pi \le x \le 2\pi \qquad (4)$$

$$\text{Arc Sin}\ (x) = x + \sum_{n=0}^{m} \frac{2x4x6\ldots\ 2n}{1x3x5\quad (2n+1)}\, x^{2n+1} \qquad 0 \le x \le .5 \qquad (5)$$

m is chosen so that the magnitude of the last term in the truncated power series (for the specific value of x) is the first term which is just less than or equal to 10^{-6}.

The following is an example of code generating for a program. The program is the same as in Figure 1.

Figure 2 shows the interaction needed between "Flow" and the user to generate his flow chart, as well as the flow-chart generated.

Figure 3 shows the communication between "Auto" and the user. "Auto" either allows the user to input his own code or to enter the information it needs to generate the code.

Figure 4 is the final code "Auto Programming" has generated. It is important to notice that the final code uses the user's variable names and their correct size, and that the labels generated by "Auto" do not conflict with that generated by the user.

3.5.3. Conclusion

The "Auto-Programming" package offers the following three advantages:

1. Ease of Operation The software package is self-explanatory. The user needs to know only how to gain access to the system. Once a connection is established, the system will ask the user for all the information needed to generate the correct program.

2. Reliability The code stored in the library will be pre-tested and debugged.

3. Time and Cost All indications are that both time and cost of writing and debugging programs are reduced. An attempt will be made this summer to substantiate the above statement. The example discussed in this paper required 10 min. of terminal time and 18 sec. of C. P. U. time.

3. 6 Application of Logical Operations to Finding Feasible Paths in a Program Flowchart -by Gary S. Popkin

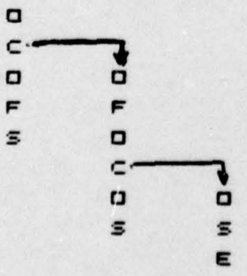
In [1], a flowchart was given in which it was desired to find all the feasible paths. The flowchart is reproduced here as Fig. 5. The matrix P, introduced in [1], displayed all feasible and infeasible paths in a flowchart. The matrix P is reproduced here as Table 1.

```
WHAT BLOCK DO YOU WANT ?
INPUT 1 FOR CONTROL BLOCK
INPUT 2 FOR A FUNCTION BLOCK
INPUT 3 FOR STOP BLOCK
INPUT 4 FOR OTHER BLOCK
INPUT 5 FOR END BLOCK
INPUT 6 FOR AN ERROR



4
1
4
2
3
4
2
4
1
4
3
4
3
```

FLOW CHART OF PROGRAM

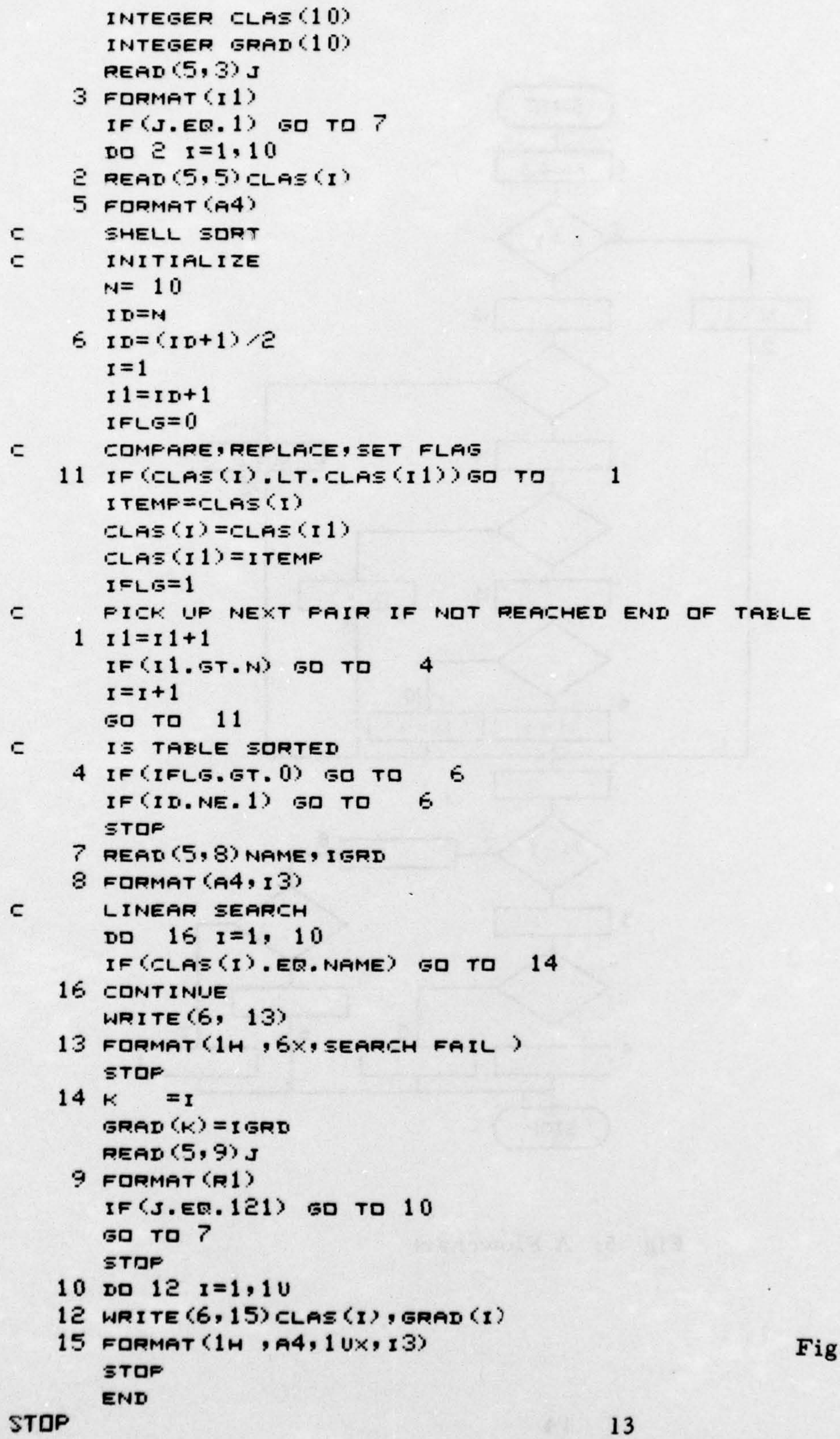


```
O
C
O         O
F         F
S         O
          C
          O         O
          S         S
                    E
```

Fig. 2: Flowchart Generation

```
EACH LINE ENDS WITH ◆
LAST LINE ENDS WITH +
      INTEGER CLAS(10)◆
      INTEGER GRAD(10)◆
      READ(5,3)J◆
    3 FORMAT(I1)+
EACH LINE ENDS WITH ◆
LAST LINE ENDS WITH +
      IF(J.EQ.1) GO TO 7+
EACH LINE ENDS WITH ◆
LAST LINE ENDS WITH +
      DO 2 I=1,10◆
    2 READ(5,5)CLAS(I)◆
    5 FORMAT(A4)+
INPUT 10 FOR SORT
INPUT 11 FOR SEARCH
10
WHICH SORT DO YOU WANT
INPUT 13 FOR INTERCHANGE SORT
INPUT 12 FOR SHELL SORT
INPUT 14 FOR BUCKET SORT
12
INPUT TABLE NAME,SIZE
CLAS 10
EACH LINE ENDS WITH ◆
LAST LINE ENDS WITH +
    7 READ (5,8)NAME,IGRD◆
    8 FORMAT(A4,I3)+
INPUT 10 FOR SORT
INPUT 11 FOR SEARCH
11
INPUT 15 FOR LINEAR SEARCH
INPUT 16 FOR BINARY SEARCH
15
INPUT TABLE NAME SIZE AND WORD BEING SEARCHED FOR, LOCATION INDEX
CLAS 10NAMEK
EACH LINE ENDS WITH ◆
LAST LINE ENDS WITH +
      GRAD(K)=IGRD◆
      READ(5,9)J◆
    9 FORMAT(R1)+
EACH LINE ENDS WITH ◆
LAST LINE ENDS WITH +
      IF(J.EQ.121) GO TO 10+
EACH LINE ENDS WITH ◆
LAST LINE ENDS WITH +
      GO TO 7+
EACH LINE ENDS WITH ◆
LAST LINE ENDS WITH +
   10 DO 12 I=1,10◆
   12 WRITE(6,15)CLAS(I),GRAD(I)◆
   15 FORMAT(1H ,A4,10X,I3)+
```

Fig. 3: Illustration of the Communication Between the Program Auto and the User



```
      INTEGER CLAS(10)
      INTEGER GRAD(10)
      READ(5,3)J
    3 FORMAT(I1)
      IF(J.EQ.1) GO TO 7
      DO 2 I=1,10
    2 READ(5,5)CLAS(I)
    5 FORMAT(A4)
C       SHELL SORT
C       INITIALIZE
      N= 10
      ID=N
    6 ID=(ID+1)/2
      I=1
      I1=ID+1
      IFLG=0
C       COMPARE,REPLACE,SET FLAG
   11 IF(CLAS(I).LT.CLAS(I1))GO TO      1
      ITEMP=CLAS(I)
      CLAS(I)=CLAS(I1)
      CLAS(I1)=ITEMP
      IFLG=1
C       PICK UP NEXT PAIR IF NOT REACHED END OF TABLE
    1 I1=I1+1
      IF(I1.GT.N) GO TO    4
      I=I+1
      GO TO   11
C       IS TABLE SORTED
    4 IF(IFLG.GT.0) GO TO    6
      IF(ID.NE.1) GO TO    6
      STOP
    7 READ(5,8)NAME,IGRD
    8 FORMAT(A4,I3)
C       LINEAR SEARCH
      DO  16 I=1, 10
      IF(CLAS(I).EQ.NAME) GO TO  14
   16 CONTINUE
      WRITE(6, 13)
   13 FORMAT(1H ,6X,SEARCH FAIL )
      STOP
   14 K    =I
      GRAD(K)=IGRD
      READ(5,9)J
    9 FORMAT(R1)
      IF(J.EQ.121) GO TO 10
      GO TO 7
      STOP
   10 DO 12 I=1,1U
   12 WRITE(6,15)CLAS(I),GRAD(I)
   15 FORMAT(1H ,A4,1UX,I3)
      STOP
      END
STOP
```

Fig. 4: The Final Code

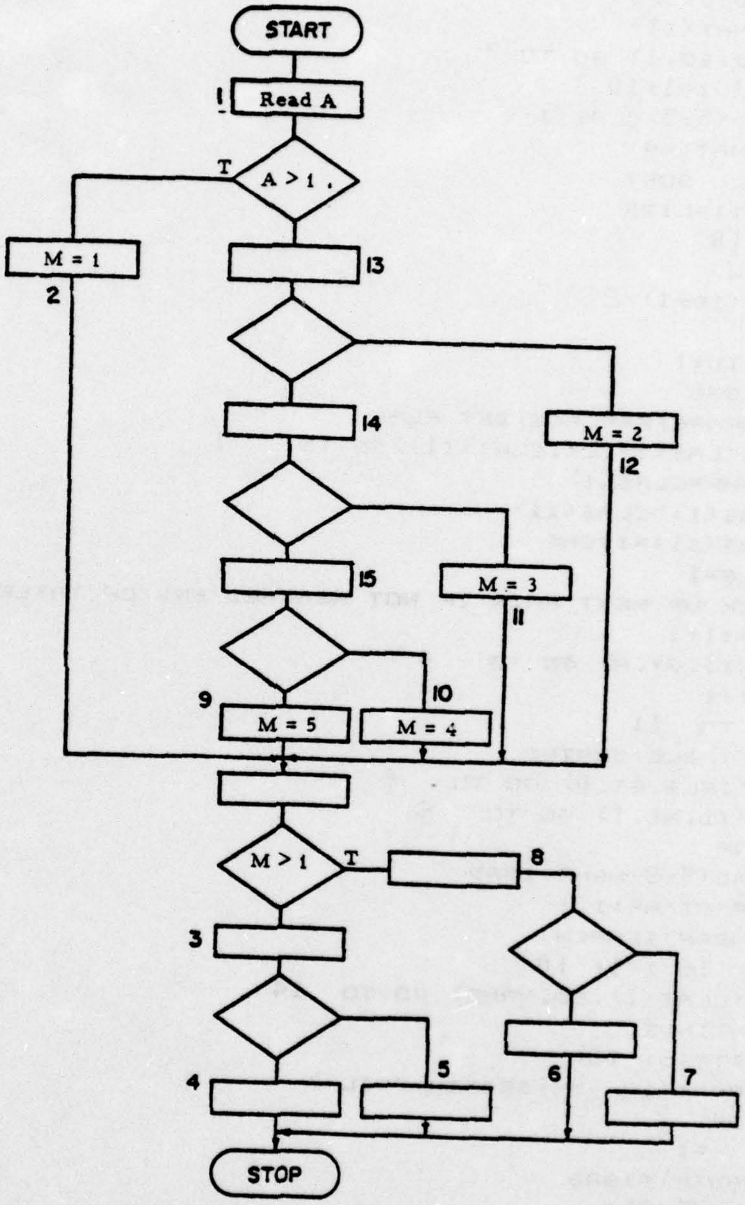


Fig. 5: A Flowchart

$$P = \begin{array}{r|cccccccccccccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 \\
\hline
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
2 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
5 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
6 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
7 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
8 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\
9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
10 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
11 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
12 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
13 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
14 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
15 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0
\end{array}$$

Table 1. The path matrix for the flowchart in Figure 5.

The flowchart in Figure 5 is small enough so that the feasible paths can be found by inspection, as was done in [1]. It is desirable to have a more methodical way of finding the feasible paths for larger flowcharts. The method proposed in [1] was to find the feasible paths by logical operations upon P, and then to remove the corresponding columns from P. If the method of finding infeasible paths was imperfect, as it promises to be, and some infeasible paths remained undetected, then the remaining matrix would contain not only all the feasible paths and but also some infeasible ones. The infeasible paths would then be found and removed by methods described in [1], as the matrix remaining from P was used in further reductions.

The approach followed for finding the columns of P (which represent the infeasible paths), was to establish vector functions of feasible paths. Subsequently, logical operations were performed between the vectors and the columns of P. For example, from the flowchart it can be seen that any feasible path containing segment 2 must also contain segment 3, and conversely. This can be expressed as a column vector C_1 whose transpose is $C_1' = (0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$. Also, any feasible path containing segment 12 must also contain segment 8, but not conversely. Perhaps this could be expressed as $C_2' = (0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0)$. In any case, the tester of the program could construct a number of such vectors describing the relationships existing the flow-chart. Unfortunately, this approach was not fruitful.

It may be possible to perform ANDing operations between the C_i and the columns of P and to find columns which represent infeasible paths.[1] For example, the following was tried and found wanting: Any column of P, say k_j, represents an infeasible path if and only if

$$C_i \circ k_j \neq C_i \qquad \text{all } i \qquad (1)$$

Using just C_1 and C_2, it so happens that (1) would correctly detect paths 3, 4, 5, and 6 as infeasible. It would incorrectly call paths 11 and 12 infeasible. (1) could be made to work for path 11 and 12 by adding to the set of the C_i a vector which says that any path containing segment 9 must also contain segment 8, namely $C'_3 = (0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0)$. But for this procedure to be workable it cannot depend on the set of the C_i being complete. While it is not fatal if the procedure leaves some infeasible paths undetected, it must not under any circumstances incorrectly call a feasible path infeasible.

In an attempt to remedy the defect involved in needing a complete set of the C_i, ANDing with single vectors C_i instead of the complete set was tried. Unfortunately this approached also failed.

Another approach would be to change the inequality in (1) to an equality and search for feasible paths instead of infeasible ones, but there still is the risk in that method that some feasible paths would be called infeasible because of the set of the C_i being incomplete.

Also, an approach using ORing was tried, with results that were even less acceptable than those from ANDing.

The above summarizes the approaches undertaken, and the difficulties that surfaced. Current effort on this problem involves: (1) further search for a satisfactory algorithm to estimate the number of feasible paths, and (2) methods for constructing an automatic test driver, where the number of inputs will equal the number of feasible paths (see Section 3. 8. 6).

REFERENCE

1. Gary S. Popkin, "Program Paths and the Number of Tests Needed to Verify a Computer Program," "Summary of Technical Progress, Software Modeling Studies," RADC-TR-76-143, pp. 40-49, May, 1976.

3.7 Statistical Theory of Program Testing and Proving - by Arthur E. Laemmel

3.7.1 Introduction

The purpose of this work is to extend the results on program testing, reported in the preceding progress report.

The usual approach to proving program correctness has so far been applied successfully only to small programs. The practical approach to large programs is to run a number of tests. The larger the number of such tests, completed without errors, the greater is our confidence in the program.

The method presented here is to combine both these approaches, this being partially theoretical program proving, and partially experimental testing.

3.7.2 A Model of Test Strategy

Observe that the result of testing a program is one of the following two outcomes:

1) The program failed on input x.

or

2) The program operates correctly on n inputs.

The same approach is often followed to investigate in a preliminary fashion the validity of a conjectured mathematical theorem. By analogy, we choose parameter values, substitute into the postulated theorem, and calculate the output. The result of this process is completely analogous to the two outcomes stated above. In a mathematical theorem, once we satified outselves that no counter example exists for a great number of tries, we may proceed and try to prove the postulated theorem. In a program we will not try to establish the validity of the program by proof, but be content with the confidence achieved after a large number of tests. In fact, M. Rabin challenges the notion of mathematical proof in the context of computational algorithms. He states that it is sufficient to establish that a statement is true to a very high probability (e.g., $1-2^{-100}$) rather than to demand an exact proof [1].

We can model the above test process and optimize it by analogy with the well-known, elementary probability example of drawing a ball from a box. We will follow the model suggested by W. L. Black[2].

A version of the model might be rephrased in traditional terms as follows: a single red ball is in one of two boxes, each of which contains a very large number of white balls. Let

p_i = a-priori probability that red ball is in box i

m_i = probability that if red ball is in box i then a single look will miss it

c_i = cost of a single look in box i

Black has given a simple way to find the minimum expected cost strategy for searching for the red ball. Arrange the numbers

$$\frac{p_i\, m_i^n\; (1-m_i)}{c_i} \quad \begin{cases} i = 1, 2 \\ n = 1, 2, 3 \ldots \end{cases}$$

in decreasing order. If the k'th number in this arrangement is one with i=1 then the k'th look should be in box i, otherwise in the other box.

In order to get a feeling for this strategy in the present application, let box 1 represent possible proofs that a certain computer program works correctly and box 2 represent various sets of input data.

Choose parameters:

$$p_1 = .9 \longleftarrow \text{(program probably OK)} \longrightarrow p_2 = .1$$

$$\left.\begin{array}{l} m_1 = .999 \\ c_1 = \;\; 100 \end{array}\right\} \begin{array}{l} \text{computer search for} \\ \text{a proof time-consuming} \\ \text{and not likely to succeed} \end{array} \qquad \left.\begin{array}{l} m_2 = .2 \\ c_2 = \;\; 5 \end{array}\right\} \begin{array}{l} \text{one} \\ \text{program} \\ \text{test is easy} \end{array}$$

$$\frac{p_1\,(1-m_1)}{c_1} = .000009 \qquad \frac{p_2(1-m_2)}{c_2} = .016$$

$$\frac{p_1\,(1-m_1)m_1}{c_1} = .00000899 \qquad \frac{p_2(1-m_2)m_2}{c_2} = .0032$$

$$\frac{p_1(1-m_1)m_1^2}{c_1} = .00000898 \qquad \frac{p_2(1-m_2)m_2^2}{c_2} = .00064$$

$$.00000897 \qquad .00128$$

$$.00000896 \qquad .0000256$$

$$.00000895 \qquad .00000512$$

According to this, looks 1, 2, 3, 4, and 5 should be in box 2, then looks 6, 7, 8 (for several hundred looks) should be in box 1. This strategy simply says to look in the box which has the highest ratio of a-posteriori probability to cost at any stage. It is intuitively satisfying that the first looks were for counter examples, and (perhaps) that the sixth look was for a proof. It is not too satisfying that these looks are to be followed by α attempts at a proof, where α is given by

$$.000009\, m_1^{\alpha} = .0000512$$

$$.57 = (1-.001)^{\alpha} \approx 1-.001\alpha$$

If the first series above decreased more rapidly, and the second series decreased more slowly, the strategy would go back and forth between proof and counter example more frequently. The main difficulty is that successive trials, either for proof or counter examples, are not satistically independent as is required in Black's model.

Each attempt at a proof can build on the last attempt because any partial results obtained in the previous attempt can be used in the next. This means that m_1 should decrease with n instead of remaining constant. On the other hand, if a computer program is tested with 10 random inputs, and if no failure occurs for the first 9, than the 10th trial certainly gives less information than the 1st trial. This means that m_2 should increase with n, i.e. that errors are harder to find with a single test later on in the testing sequence. Such a result can be derived from formulas given in the previous report, since these represent a way to describe statistical dependence among test outcomes.

Incidentally, the same approach was successfully applied to the optimization of a reliability structure under cost constraint by M. Messinger and M. L. Shooman[3]. The algorithm allocated redundant components to the structure in a sequence in such a way that each addition of a component maximized the gain in reliability per dollar.

3.7.3 Conclusion

The work is continuing in this area. The object is to formalize the interactive part-proof, part-test procedure to program verification with a prescribed confidence level.

References

1. M. Rabin, Carnegie-Mellon Symposium on New Directions and Recent Results in Algorithms and Complexity, April 7-9, (reported in Paper Crisis, SIAM News, Vol. 9, number 4, August 1976).

2. W. L. Black, "Discrete Sequential Search" Information and Control, vol. 8, pp. 159-162, 1965.

3. M. Messinger and M. Shooman, "Optimum Allocation of Spares Redundancy: A Tutorial Survey." IEEE Transactions on Reliability, July 1071.

3.8 Implementation of Shooman's Model of Exhaustive Testing - An Automatic Type 1.A Tester - by Denis L. Baggi

3.8.1 Introduction

In an internal paper "Analytical Models for Software Testings" Martin L. Shooman describes a scheme for implementing a driver program to automatically test each path of a given program. An implementation of this scheme in PL/1, with a few revisions, is described here, along with two examples of programs, which were run with normal analytical debugging techniques - i.e., with some testing data - and through the testing program. Comparisons among man-hour efforts and computer time in both cases are made.

3.8.2 The Testing Driver Program

A program, referred to as driver program, has been developed and run in conjunction with two programming examples. Its purpose is to allow automatic testing of all possible paths of any given program. A description of its functioning follows.

The driver program requires a data card containing an integer, N-TESTS, i.e., the number of IF statements, plus the number of repetitive DO groups, in the program and subroutines, to be supplied by the programmer.

The next data item has to be an "order," i.e., a character string such as 'NORMAL OPERATION,' or 'TEST,' or any other string. If the order is 'NORMAL OPERATION,' then the driver allows normal functioning of the program to be tested, e.g., with a set of data designed, by the programmer, to test some cases - a normal debugging practice.

If the order is 'TEST,' no data set is needed for the tested program, but an array, T, with lower bound 1 and upper bound N-TESTS (the number of tests, as read in previously), will be constructed to represent, in ascending order, all possible bit combinations of binary numbers from 0 up to 2 ** N-TESTS - 1; this array is called testing word, and it thus consists of the bits of a binary counter with N-TESTS bits. Notice that the value 0 is in fact represented, in the corresponding T, by - 1, while 1 is represented by 1. Eventually the program to test is run for any such binary combination.

For any other order string, such as 'ENOUGH,' as well as in the case of absence of data, the whole system stops.

3.8.3. The Tested Program

Although no particular care has been taken to make sure that the driver program is fully compatible with all possible programs to test it is believed that, at its present state, the invariant part is flexible enough to accept a large class of programs with no modification, requiring for other programs only minor, sensible changes.

Shooman indicates (in p. 5-1 of his paper), a strategy for implementing the driver program, namely, a revised way of writing IF statements and DO loops in a program to be submitted to testing; however, since such schemes lack generality (-i.e., only conditions of the type "exp > 0" are allowed in IF statements, and only limits from 1 up to an upper bound > 0 in DOloops -), the scheme described here has been developed as a natural derivative of these suggestions; hence, the only restrictions to be obeyed in writing a program will be the following:

1a) instead of IF cond THEN statement$_1$; ELSE statement$_2$;

write IF F(cond) THEN statement$_1$; ELSE statement$_2$;

1b) instead of DO I = LIMIT 1 TO LIMIT 2 BY INCR ;

write DO I = GL (LIMIT 1, LIMIT 2) TO GH BY INCR;

1c) instead of DO WHILE (cond) ;

write DO WHILE (H(cond)) ;

(where: F, GL, GH and H are described in the forth coming technical report)

2) function and subroutine procedures are possible but should be internal to the program

3) variables used in the program which are assigned a value through a read (GET LIST) statement should be initialized, for instance through a DCL INIT statement.

All these restrictions could be removed. To remove 1), one could construct a subprogram in the operating system which automatically supplies F, GL and GH, and H. Subroutines could be external, as long as a mechanism is provided for passing back and forth T - e.g., COMMON statements in FORTRAN -, hence removing 2). And point 3) is a direct consequence of the read-in scheme described by Shooman, which could be modified at will.

3.8.4 The Two Examples

For illustration two programs were chosen at the two ends of a spectrum: one with many IF statements, and input data, and the other with DO groups and one subroutine, and no input data.

A. First Example : Computer Solution of a Card Game

This is a very slightly altered version of Shooman's algorithm of Fig. 4-3 in his paper. The algorithm appears in Fig. 8. It determines the winner of a card game, in which player A is dealt two cards, A1 and A2, and B, similarly, gets B1 and B2, (four integers read in with a GET LIST statement). If both winners have a pair, the highest pair wins, or if they are equal it is

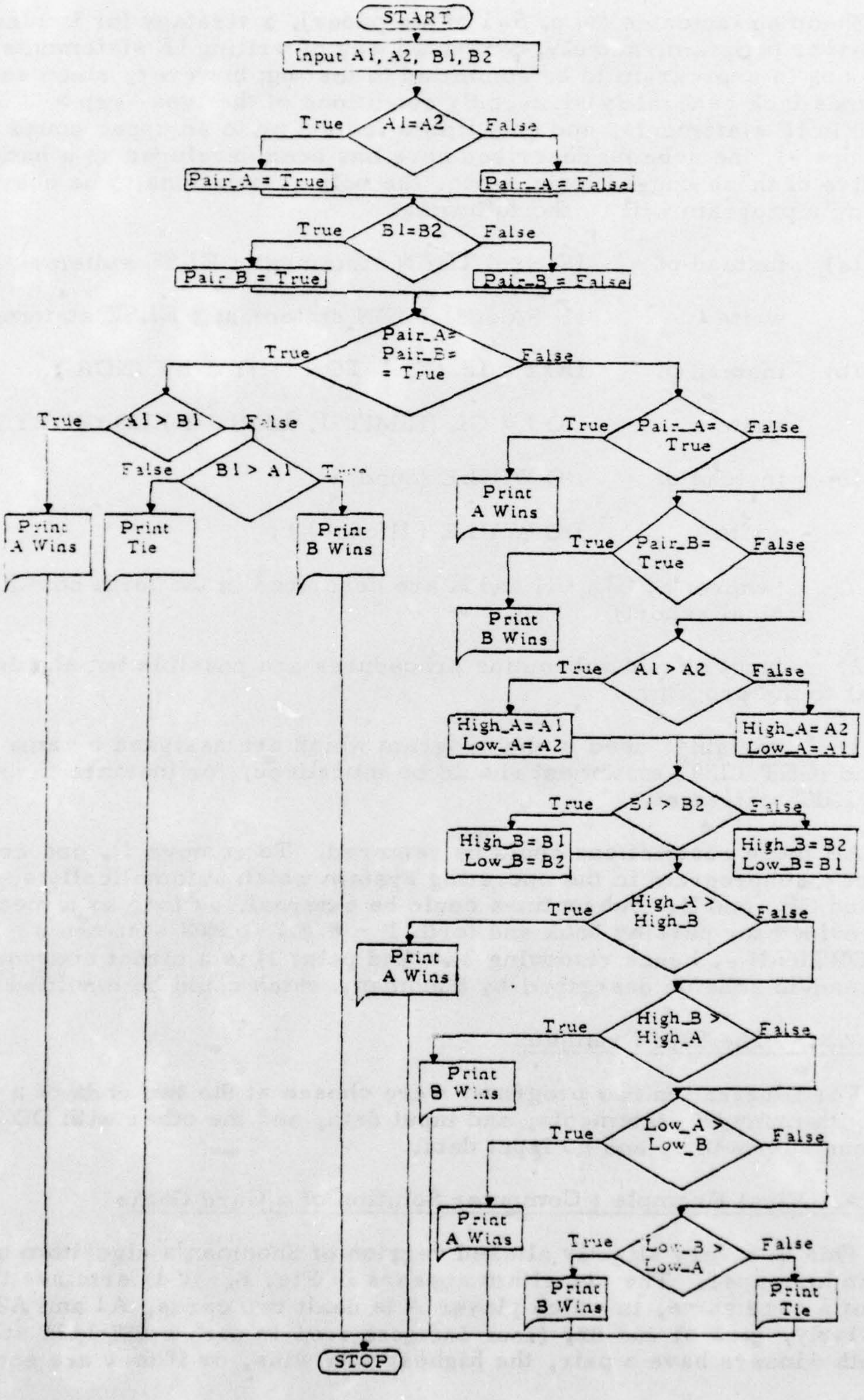


Fig. 8: Flow Chart for Computer Solution of a Card Game

a tie; if only one player has a pair, he wins; otherwise the highest card wins, or if they are equal, the highest second card wins; identical hands are ties. At first the system was run under 'NORMAL OPERATION' conditions. The results are shown in the forth coming report. The system was next run under 'TEST' conditions. The tested program has 12 IF statements, hence we have a 12-bit testing word.

Since the program is fully debugged, no error can be seen in the output listing; should one error appear, however, it would be easy, from the testing word, to reconstruct the path and detect the deficient statement.

B. Second example: A Program Which Prints the Prime Factors of all Integers from 1 to 100

This is a simple program which tests each integer from 1 to 100, prints it, and its prime factors, or the word PRIME if it is prime. Its algorithm is presented in Fig. 9.

The internal subroutine PRINTOUT prints the results; this procedure has been incorporated to show the generality of the scheme including subprograms. Notice that, although it is called only once from the main program, it could be invoked as many times as needed, because of the design of the internal procedures of the driver program, which know by themselves how to select a new bit in the testing word each time they are used.

3.8.5 Efficiency of the System

Program Example A.

- it took 30 minutes to design the program
- it takes no extra time to redesign a program according to the specifications expressed in section 3.8.3
- it took 10 minutes to find a data set to test some well-choosen paths
- the program ran in 2.41 minutes under 'NORMAL OPERATION' with that set of data
- the program ran in 4.12 minutes under 'TEST' conditions, exploring all paths

Program Example B

- it took 20 minutes to design the program
- there is no data
- the program ran in 3.81 minutes for 100 integers
- the program ran in 1.76 minutes under TEST conditions

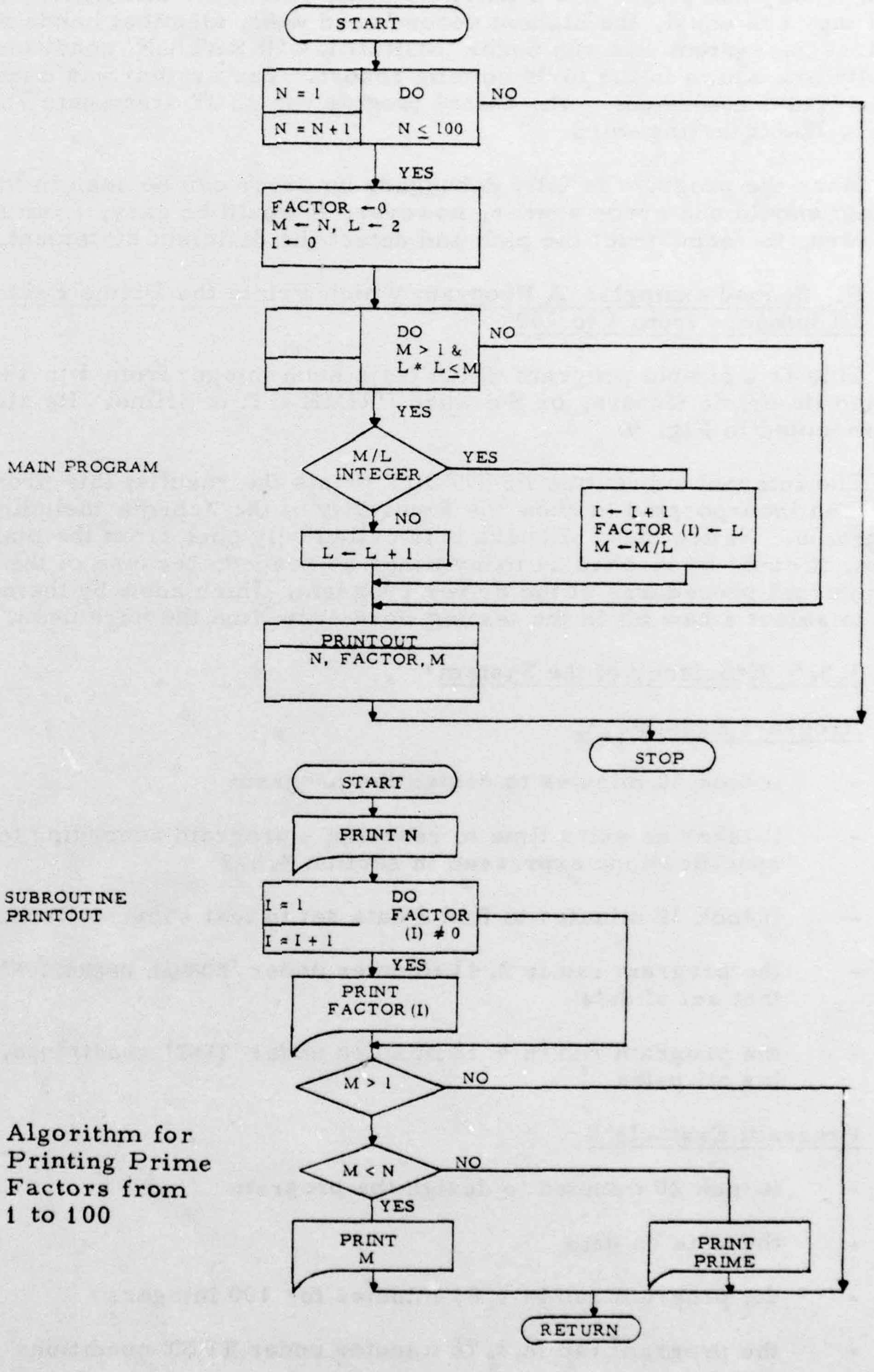


Fig. 9: Algorithm for Printing Prime Factors from 1 to 100

Hence the system provides the following:

Advantage

No time is spent in finding a data set to debug a program.

Disadvantage

Running time may increase exponentially with the number of IF statements and DO loops. For instance, Program A contains 12 IF statements, and therefore the testing word has 13 bits and 8192 runs through the program were needed to test its 100 paths (hence, with this blind mechanical approach many tests are meaningless). However, if the program becomes too large, it can be separated in portions to be tested individually, along with the interconnecting data sets. Furthermore, this disadvantage is compensated by those cases of programs with many DO-loops and few IF statements. Program B, for instance, required almost four minutes for a hundred integers, and would use a lot more for, say 10,000 integers, but it took less than two minutes to go through all paths as defined, and would still take the same amount of time no matter how many integers it would have to consider. Hence a TEST run is, in these cases, very time saving.

3.8.6 Conclusions

A possible implementation for an automatic program tester of type 1.A has been discussed. The tester ignores the semantics of the tested program, but is able however to run through all possible paths present in a program and catch a possible error.

This implementation, rather than representing an ultimate result, is meant to be an illustration of the method and techniques proposed by Shooman in his paper; it would be easy, for instance, to make the driver program more flexible or suited to other styles or programming languages.

It was rewarding to discover that even within the limited development of these techniques some goals have been achieved, namely, the realization of a system which has already proved its usefulness in debugging the described programs.

4.0 Directions for Next Period's Work

In the next period we plan to work on the following:

1. **Adamowicz:** Further work on measures for the evaluation of software.

2. **Baggi:** Completion of a report on the construction of an automatic driver for Shooman's model of test covering each program. path.

3. **Laemmel:** Continuation of studies on statistical program testing.

4. **Marshall:** Application of graph theory to statistical sampling approaches for software reliability.

5. **Ruston and Berlinger:** Applications of software physics to complexity measures.

6. **Shooman and Popkin:** Continuation of work on levels of program testing

7. **Shooman and Ruston :** Experimental tests for (1) validation of seeding/tagging estimates (2) Shooman's extended debugging models (incorporating errors generated during the debugging process), and (3) obtaining data for verification of software physics and other complexity measures.

5.0 Professional Activities

This section summarizes the professional activities of the research personnel working on this contract.

5.1 Published and Submitted Papers and Reports

1. C. L. Hsu and L. Shaw, "Downtime Distributions Based on a Multivariate Exponential Distribution," Report No. Poly EE/EP 76-002 EER 120, Polytechnic Institute of New York, Feb. 1976.

2. C. Marshall, "Contributions to the Theory of Avaiilability," Report No. Poly EE/EP 76-004 EER 121, Polytechnic Institute of New York, Feb. 1976.

3. S. N. Mohanty and M. Adamowicz, "Proposed Measures for the Evaluation of Software," to appear with Proceedings of the Symposium on Computer Software Engineering, 1976.

4. L. Shaw and M. Shooman, "Confidence Bounds and Propagation of Uncertainties in System Availability and Reliability Computations," Technical Report N00014-67-A-0438-0013, Poly EE/EP 75-002 Polytechnic Institute of New York, Jan. 1976.

5. L. Shaw and S. Sinkar, "Redundant Spares Allocation to Reduce Reliability Costs," Naval Research Logistics Quarterly vol. 23, No. 2, pp. 179-194, June 1976.

6. M. L. Shooman, "Recent Developments in Software Reliability - The State of the Art," To appear in the Proceedings of the Thirteenth IEEE Computer Society International Conference, Washington, D. C., Sept. 1976.

7. M. L. Shooman "Structural Models for Software Reliability Prediction," Second National Conference on Software Engineering, October, 1976, San Francisco, Calif.

8. M. L. Shooman and M. I. Bolsky, "Types, Distribution, and Test Correction Times for Programming Errors," IEEE Transactions on Reliability, vol. R-25, No. 2, pp. 69-70, June 1976.

9. M. L. Shooman, M. Horodniceanu, and E. J. Cantilli, "System Safety Applied to Transportation Systems," To appear in the Proceedings of Inter Society Conference on Transportation, Los Angeles, Calif. July 1976.

10. M. L. Shooman and S. Natarajan, "Effect of Manpower Depolyment and Error Generation on Software Reliability," to appear in the Proceedings of the Symposium on Computer Software Engineering, 1976.

11. M. L. Shooman and H. Ruston, "Cost Reducing, High Reliability Programming Techniques," accepted for the 1976 ORSA/TIMS Joint National Meeting, November, 1976.

12. M. L. Shooman and S. Sinkar, "Generation of Reliability and Safety Data by Analysis of Expert Opinion," accepted for the 1977 Annual Reliability and Maintainability Symposium, Philadelphia, PA.

13. M. L. Shooman and A. K. Trivedi, "A Many-State Markov Model for Computer Software Performance Parameters," IEEE Transactions on Reliability, vol. R-25, No. 2, pp. 66-68, June 1976.

5.2 Talks and Seminars

1. S. Habib, "An Overview of Microprocessors," Seminar, PINY, February 1976.

2. H. Ruston, "Top-down Design," Computer Seminar, PINY, March 1976.

3. D. Baggi, "Design of Automatic Test Drivers," Seminar, PINY, June 1976.

4. M. L. Shooman, H. Ruston, A. Sukert, E. Berlinger A. Laemmel, E. Lipshitz C. Marshall, B. Rudner, "Software Engineering Topics," Oral Presentation of Progress on Studies Supported by the RADC Program, PINY, June 1976.

5. S. Habib, "User Services in Remote Entry Environment," National Science Foundation Conference on Computers in Undergraduate Education, Binghamton, NY June 1976.

5.3 Symposia and Technical Societies

1. M. L. Shooman, Chairman, Program Committee, MRI Symposium on Computer Software Engineering, New York City, April 1976.

2. M. Adamowicz, S. Habib, A. Laemmel and H. Ruston, Members, Program Committee, MRI Symposium on Computer Software Engineering.

5.4 Committees

1. M. L. Shooman, Member, IEEE ADCOM (Administrative Committee) of the Group on Reliability.

2. M. L. Shooman, Member, Executive Committee, IEEE Computer Society Technical Committee on Software Engineering.

3. M. L. Shooman, Member, NASA Advisory Committee on Guidance, Control and Information Systems.

4. S. Habib, Chairman, National Lectureship Committee of the Association for Computing Machinery.

5. S. Habib, Chairman, SIGMICRO (Sepical Interest Group on Microprogramming) of ACM.

METRIC SYSTEM

BASE UNITS:

Quantity	Unit	SI Symbol	Formula
length	metre	m	...
mass	kilogram	kg	...
time	second	s	...
electric current	ampere	A	...
thermodynamic temperature	kelvin	K	...
amount of substance	mole	mol	...
luminous-intensity	candela	cd	...

SUPPLEMENTARY UNITS:

Quantity	Unit	SI Symbol	Formula
plane angle	radian	rad	...
solid angle	steradian	sr	...

DERIVED UNITS:

Quantity	Unit	SI Symbol	Formula
Acceleration	metre per second squared	...	m/s
activity (of a radioactive source)	disintegration per second	...	(disintegration)/s
angular acceleration	radian per second squared	...	rad/s
angular velocity	radian per second	...	rad/s
area	square metre	...	m
density	kilogram per cubic metre	...	kg/m
electric capacitance	farad	F	A·s/V
electrical conductance	siemens	S	A/V
electric field strength	volt per metre	...	V/m
electric inductance	henry	H	V·s/A
electric potential difference	volt	V	W/A
electric resistance	ohm	...	V/A
electromotive force	volt	V	W/A
energy	joule	J	N·m
entropy	joule per kelvin	...	J/K
force	newton	N	kg·m/s
frequency	hertz	Hz	(cycle)/s
illuminance	lux	lx	lm/m
luminance	candela per square metre	...	cd/m
luminous flux	lumen	lm	cd·sr
magnetic field strength	ampere per metre	...	A/m
magnetic flux	weber	Wb	V·s
magnetic flux density	tesla	T	Wb/m
magnetomotive force	ampere	A	...
power	watt	W	J/s
pressure	pascal	Pa	N/m
quantity of electricity	coulomb	C	A·s
quantity of heat	joule	J	N·m
radiant intensity	watt per steradian	...	W/s.
specific heat	joule per kilogram-kelvin	...	J/kg·K
stress	pascal	Pa	N/m
thermal conductivity	watt per metre-kelvin	...	W/m·K
velocity	metre per second	...	m/s
viscosity, dynamic	pascal-second	...	Pa·s
viscosity, kinematic	square metre per second	...	m/s
voltage	volt	V	W/A
volume	cubic metre	...	m
wavenumber	reciprocal metre	...	(wave)/m
work	joule	J	N·m

SI PREFIXES:

Multiplication Factors	Prefix	SI Symbol
1 000 000 000 000 = 10^{12}	tera	T
1 000 000 000 = 10^{9}	giga	G
1 000 000 = 10^{6}	mega	M
1 000 = 10^{3}	kilo	k
100 = 10^{2}	hecto*	h
10 = 10^{1}	deka*	da
0.1 = 10^{-1}	deci*	d
0.01 = 10^{-2}	centi*	c
0.001 = 10^{-3}	milli	m
0.000 001 = 10^{-6}	micro	μ
0.000 000 001 = 10^{-9}	nano	n
0.000 000 000 001 = 10^{-12}	pico	p
0.000 000 000 000 001 = 10^{-15}	femto	f
0.000 000 000 000 000 001 = 10^{-18}	atto	a

* To be avoided where possible.

MISSION
of
Rome Air Development Center

RADC plans and conducts research, exploratory and advanced development programs in command, control, and communications (C^3) activities, and in the C^3 areas of information sciences and intelligence. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

AMERICAN REVOLUTION BICENTENNIAL
1776-1976

FILM

4